方法（Methods）

1.0 翻译：pp-prog (https://github.com/pp-prog) 校对：zqp (https://github.com/zqp)

2.0 翻译+校对：DianQK (https://github.com/DianQK)

2.1 翻译：DianQK (https://github.com/DianQK)，Realank (https://github.com/Realank) 校对：shanks (http://codebuild.me)，2016-01-18

2.2 校对：SketchK (https://github.com/SketchK) 2016-05-13

本页包含内容：

方法是与某些特定类型相关联的函数。类、结构体、枚举都可以定义实例方法；实例方法为给定类型的实例封装了具体的任务与功能。类、结构体、枚举也可以定义类型方法；类型方法与类型本身相关联。类型方法与 Objective-C 中的类方法（class methods）相似。

结构体和枚举能够定义方法是 Swift 与 C/Objective-C 的主要区别之一。在 Objective-C 中，类是唯一能定义方法的类型。但在 Swift 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活地在你创建的类型（类/结构体/枚举）上定义方法。

实例方法 (Instance Methods)

实例方法是属于某个特定类、结构体或者枚举类型实例的方法。实例方法提供访问和修改实例属性的方法或提供与实例目的相关的功能，并以此来支撑实例的功能。实例方法的语法与函数完全一致，详情参见函数 (./06_Functions.html)。

实例方法要写在它所属的类型的前后大括号之间。实例方法能够隐式访问它所属类型的所有的其他实例方法和属性。实例方法只能被它所属的类的某个特定实例调用。实例方法不能脱离于现存的实例而被调用。

下面的例子，定义一个很简单的 `Counter` 类， `Counter` 能被用来对一个动作发生的次数进行计数：

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

`Counter` 类定义了三个实例方法：

- `increment` 让计数器按一递增；
- `incrementBy(amount: Int)` 让计数器按一个指定的整数值递增；
- `reset` 将计数器重置为0。

`Counter` 这个类还声明了一个可变属性 `count` ，用它来保持对当前计数器值的追踪。

和调用属性一样，用点语法（dot syntax）调用实例方法：

```
let counter = Counter()
// 初始计数值是0
counter.increment()
// 计数值现在是1
counter.incrementBy(5)
// 计数值现在是6
counter.reset()
// 计数值现在是0
```

方法的局部参数名称和外部参数名称 (Local and External Parameter Names for Methods)

函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用），详情参见指定外部参数名(./06_Functions.html#specifying_external_parameter_names)。方法参数也一样（因为方法就是函数，只是这个函数与某个类型相关联了）。

Swift 中的方法和 Objective-C 中的方法极其相似。像在 Objective-C 中一样，Swift 中方法的名称通常用一个介词指向方法的第一个参数，比如： `with`， `for`， `by` 等等。前面的 `Counter` 类的例子中 `incrementBy(_:)` 方法就是这样的。介词的使用让方法在被调用时能像一个句子一样被解读。

具体来说，Swift 默认仅给方法的第一个参数名称一个局部参数名称；默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。这个约定与典型的命名和调用约定相适应，与你在写 Objective-C 的方法时很相似。这个约定还让富于表达性的方法在调用时不需要再限定参数名称。

看看下面这个 `Counter` 的另一个版本（它定义了一个更复杂的 `incrementBy(_:)` 方法）：

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes: Int) {
        count += amount * numberOfTimes
    }
}
```

`incrementBy(_:numberOfTimes:)` 方法有两个参数： `amount` 和 `numberOfTimes` 。默认情况下，Swift 只把 `amount` 当作一个局部名称，但是把 `numberOfTimes` 即看作局部名称又看作外部名称。下面调用这个方法：

```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// counter 的值现在是 15
```

你不必为第一个参数值再定义一个外部变量名：因为从函数名 `incrementBy(_numberOfTimes:)` 已经能很清楚地看出它的作用。但是第二个参数，就要被一个外部参数名称所限定，以便在方法被调用时明确它的作用。

上面描述的这种默认行为意味着在 Swift 中，定义方法使用了与 Objective-C 同样的语法风格，并且方法将以自然且富于表达性的方式被调用。

修改方法的外部参数名称(Modifying External Parameter Name Behavior for Methods)

有时为方法的第一个参数提供一个外部参数名称是非常有用的，尽管这不是默认的行为。你自己可以为第一个参数添加一个显式的外部名称。

相反，如果你不想为方法的第二个及后续的参数提供一个外部名称，可以通过使用下划线（ `_` ）作为该参数的显式外部名称，这样做将覆盖默认行为。

self 属性(The self Property)

类型的每一个实例都有一个隐含属性叫做 `self`， `self` 完全等同于该实例本身。你可以在一个实例的实例方法中使用这个隐含的 `self` 属性来引用当前实例。

上面例子中的 `increment` 方法还可以这样写：

```
func increment() {
    self.count += 1
}
```

实际上，你不必在你的代码里面经常写 `self` 。不论何时，只要在一个方法中使用一个已知的属性或者方法名称，如果你没有明确地写 `self`，Swift 假定你是指当前实例的属性或者方法。这种假定在上面的 `Counter` 中已经示范了： `Counter` 中的三个实例方法中都使用的是 `count`（而不是 `self.count` ）。

使用这条规则的主要场景是实例方法的某个参数名称与实例的某个属性名称相同的时候。在这种情况下，参数名称享有优先权，并且在引用属性时必须使用一种更严格的方式。这时你可以使用 `self` 属性来区分参数名称和属性名称。

下面的例子中， `self` 消除方法参数 `x` 和实例属性 `x` 之间的歧义：

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
    print("This point is to the right of the line where x == 1.0")
}
// 打印输出: This point is to the right of the line where x == 1.0
```

如果不使用 `self` 前缀，Swift 就认为两次使用的 `x` 都指的是名称为 `x` 的函数参数。

在实例方法中修改值类型(Modifying Value Types from Within Instance Methods)

结构体和枚举是**值类型**。默认情况下，值类型的属性不能在它的实例方法中被修改。

但是，如果你确实需要在某个特定的方法中修改结构体或者枚举的属性，你可以为这个方法选择 可变(`mutating`) 行为，然后就可以从其方法内部改变它的属性；并且这个方法做的任何改变都会在方法执行结束时写回到原始结构中。方法还可以给它隐含的 `self` 属性赋予一个全新的实例，这个新实例在方法结束时会替换现存实例。

要使用 可变 方法，将关键字 `mutating` 放到方法的 `func` 关键字之前就可以了：

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// 打印输出: "The point is now at (3.0, 4.0)"
```

上面的 `Point` 结构体定义了一个可变方法 `moveByX(_:y:)` 来移动 `Point` 实例到给定的位置。该方法被调用时修改了这个点，而不是返回一个新的点。方法定义时加上了 `mutating` 关键字，从而允许修改属性。

注意，不能在结构体类型的常量（a constant of structure type）上调用可变方法，因为其属性不能被改变，即使属性是变量属性，详情参见常量结构体的存储属性 (./10_Properties.html#stored_properties_of_constant_structure_instances)：

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// 这里将会报告一个错误
```

在可变方法中给 self 赋值(Assigning to self Within a Mutating Method)

可变方法能够赋给隐含属性 `self` 一个全新的实例。上面 `Point` 的例子可以用下面的方式改写：

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

新版的可变方法 `moveByX(_:y:)` 创建了一个新的结构体实例，它的 `x` 和 `y` 的值都被设定为目标值。调用这个版本的方法和调用上个版本的最终结果是一样的。

枚举的可变方法可以把 `self` 设置为同一枚举类型中不同的成员：

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
        case Off:
            self = Low
        case Low:
            self = High
        case High:
            self = Off
        }
    }
}
var ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight 现在等于 .High
ovenLight.next()
// ovenLight 现在等于 .Off
```

上面的例子中定义了一个三态开关的枚举。每次调用 `next()` 方法时，开关在不同的电源状态（ `Off` ， `Low` ， `High` ）之间循环切换。

类型方法 (Type Methods)

实例方法是被某个类型的实例调用的方法。你也可以定义在类型本身上调用的方法，这种方法就叫做**类型方法**（Type Methods）。在方法的 `func` 关键字之前加上关键字 `static` ，来指定类型方法。类还可以用关键字 `class` 来允许子类重写父类的方法实现。

> 注意
> 在 Objective-C 中，你只能为 Objective-C 的类类型（classes）定义类型方法（type-level methods）。在 Swift 中，你可以为所有的类、结构体和枚举定义类型方法。每一个类型方法都被它所支持的类型显式包含。

类型方法和实例方法一样用点语法调用。但是，你是在类型上调用这个方法，而不是在实例上调用。下面是如何在 `SomeClass` 类上调用类型方法的例子：

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

在类型方法的方法体（body）中， `self` 指向这个类型本身，而不是类型的某个实例。这意味着你可以用 `self` 来消除类型属性和类型方法参数之间的歧义（类似于我们在前面处理实例属性和实例方法参数时做的那样）。

一般来说，在类型方法的方法体中，任何未限定的方法和属性名称，可以被本类中其他的类型方法和类型属性引用。一个类型方法可以直接通过类型方法的名称调用本类中的其它类型方法，而无需在方法名称前面加上类型名称。类似地，在结构体和枚举中，也能够直接通过类型属性的名称访问本类中的类型属性，而不需要前面加上类型名称。

下面的例子定义了一个名为 `LevelTracker` 结构体。它监测玩家的游戏发展情况（游戏的不同层次或阶段）。这是一个单人游戏，但也可以存储多个玩家在同一设备上的游戏信息。

游戏初始时，所有的游戏等级（除了等级 1）都被锁定。每次有玩家完成一个等级，这个等级就对这个设备上的所有玩家解锁。 `LevelTracker` 结构体用类型属性和方法监测游戏的哪个等级已经被解锁。它还监测每个玩家的当前等级。

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel { highestUnlockedLevel = level }
    }
    static func levelIsUnlocked(level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
    var currentLevel = 1
    mutating func advanceToLevel(level: Int) -> Bool {
        if LevelTracker.levelIsUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

`LevelTracker` 监测玩家已解锁的最高等级。这个值被存储在类型属性 `highestUnlockedLevel` 中。

`LevelTracker` 还定义了两个类型方法与 `highestUnlockedLevel` 配合工作。第一个类型方法是 `unlockLevel` ，一旦新等级被解锁，它会更新 `highestUnlockedLevel` 的值。第二个类型方法是 `levelIsUnlocked` ，如果某个给定的等级已经被解锁，它将返回 `true` 。（注意，尽管我们没有使用类似 `LevelTracker.highestUnlockedLevel` 的写法，这个类型方法还是能够访问类型属性 `highestUnlockedLevel` ）

除了类型属性和类型方法， `LevelTracker` 还监测每个玩家的进度。它用实例属性 `currentLevel` 来监测每个玩家当前的等级。

为了便于管理 `currentLevel` 属性， `LevelTracker` 定义了实例方法 `advanceToLevel` 。这个方法会在更新 `currentLevel` 之前检查所请求的新等级是否已经解锁。 `advanceToLevel` 方法返回布尔值以指示是否能够设置 `currentLevel` 。

下面， `Player` 类使用 `LevelTracker` 来监测和更新每个玩家的发展进度：

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func completedLevel(level: Int) {
        LevelTracker.unlockLevel(level + 1)
        tracker.advanceToLevel(level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

`Player` 类创建一个新的 `LevelTracker` 实例来监测这个用户的进度。它提供了 `completedLevel` 方法，一旦玩家完成某个指定等级就调用它。这个方法为所有玩家解锁下一等级，并且将当前玩家的进度更新为下一等级。（我们忽略了 `advanceToLevel` 返回的布尔值，因为之前调用 `LevelTracker.unlockLevel` 时就知道了这个等级已经被解锁了）。

你还可以为一个新的玩家创建一个 `Player` 的实例，然后看这个玩家完成等级一时发生了什么：

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// 打印输出：highest unlocked level is now 2
```

如果你创建了第二个玩家，并尝试让他开始一个没有被任何玩家解锁的等级，那么试图设置玩家当前等级将会失败：

```
player = Player(name: "Beto")
if player.tracker.advanceToLevel(6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// 打印输出：level 6 has not yet been unlocked
```

上一篇：属性 (/project/swift/chapter2/10_Properties.html)

下一篇：下标 (/project/swift/chapter2/12_Subscripts.html)

2月25日 回复 顶(1) 转发


(http://weibo.com/1403384842)

悉毅 (http://weibo.com/1403384842)

回复 悉毅: 我搞错了，是实例方法中不可以更改。没有mutating的话不可以修改值类型的x和y。

1月28日 回复 顶 转发


(http://weibo.com/1403384842)

悉毅 (http://weibo.com/1403384842)

“结构体和枚举是值类型。默认情况下，值类型的属性不能在它的实例方法中被修改。”

可是我试着用：

somePoint.x += 100
somePoint.y += 104

修改实例的属性的值，系统没有报错，且容许我这么改了。

1月28日 回复 顶 转发


(http://weibo.com/wenbokenet)

dong (http://weibo.com/wenbokenet)

mutating 最好翻译成 可变 ，变异是生物意义上的，希望能采纳，(*^__^*) 嘻嘻……，辛苦了

1月4日 回复 顶 转发


(http://weibo.com/caoping)

caoping (http://weibo.com/caoping)

枚举有方法还可以理解，结构体有这么多类的特性真是不知道要干嘛，其实看举的那些结构体的例子，明显用类更好。

2015年12月8日 回复 顶 转发


(http://t.qq.com/congjinyih6382)

从今以后 (http://t.qq.com/congjinyih6382)

回复 zyg: 已经更正了 

2015年11月16日 回复 顶 转发

从今以后

回复 野鹤: 这个[类型本身]可以指 类，结构体，或者枚举。

2015年11月16日 回复 顶 转发

(http://t.qq.com/flowerfeng1987)

Howie (http://t.qq.com/flowerfeng1987)

回复 野鹤: 在 Swift 中，你可以为所有的类、结构体和枚举定义类型方法：每一个类型方法都被它所支持的类型显式包含。

2015年10月30日 回复 顶 转发


(http://t.qq.com/simadi)

野鹤 (http://t.qq.com/simadi)

你也可以定义类型本身调用的方法，这种方法就叫做类型方法.
这个[类型本身]指的是 类本身吗?还是指什么?

2015年10月28日 回复 顶 转发

壮志凌云

回复 壮志凌云: 抱歉，看错了

2015年8月19日 回复 顶 转发

壮志凌云

发现这一节有很多内容跟原文无法对应

2015年8月19日 回复 顶 转发

壮志凌云

回复 zyg: 恩，2.0中已经没有这个#号了，会尽快更正的。

2015年8月17日 回复 顶 转发

zyg

Modifying External Parameter Name Behavior for Methods 中 # 已经移出了

2015年8月11日 回复 顶 转发